RELATIONAL ALGEBRAIC OPERATIONS FOR THE IFS/2

N.E.J. Dewhurst and S.H. Lavington

Abstract: The IFS/2 is an add-on hardware unit which provides bulk operations on structures such as sets, relations and graphs. This report, which is a sequel to CSM-164, describes seven IFS/2 procedures which perform operations on relations held in the IFS/2’s persistent memory.

Internal report CSM-168, February 1992

Department of Computer Science
University of Essex
Colchester CO4 3SQ
UK

email: lavington@uk.ac.essex
Relational Algebraic Operations for the IFS/2

N.E.J. Dewhurst and S.H. Lavington*

Dept of Computer Science, University of Essex
Colchester, Essex CO4 3SQ, UK

*email: lavington@essex.ac.uk

Abstract: The IFS/2 is an add-on hardware unit which provides bulk operations on structures such as sets, relations and graphs. This report, which is a sequel to CSM-164, describes seven IFS/2 procedures which perform operations on relations held in the IFS/2's persistent memory.

1. Introduction

The IFS/2 is an add-on hardware unit which is being designed and built by the Intelligent File Store group at the University of Essex [1,2]. The IFS/2 provides support for knowledge-based systems by both storing and manipulating persistent structures such as sets, relations and graphs. The IFS/2 has a low-level procedural interface, available to programmers as a collection of C library procedures. This collection defines the IFS/2's External Procedural Interface (EPI).

A software simulator, known as SIERRA, has been written for the IFS/2. SIERRA is a suite of C procedures and associated data structures that gives, via the same EPI command structure, the same functionality as the actual IFS/2 hardware. The basic SIERRA procedures such as insert, search, and delete, are described in [3]. This present Report describes seven additional procedures which realize the IFS/2 relational algebraic operations - thus constituting an extension to the earlier version of EPI described in [3].

Readers familiar with the IFS/2 and SIERRA should now skip to Section 3. For those new to the ideas, Section 2 gives a brief review of IFS/2 terminology. Further details are contained in [3], which should be regarded as a companion report.

2. Review of IFS terminology

The IFS/2 operates according to the active memory principle, in which whole-structure operations are performed by composed commands. Since the IFS/2 has a one-level associative (ie content-addressable) memory, stored objects of various granularities may be identified and accessed by logical name or logical descriptor, rather than by physical address.

The IFS/2's basic unit of information is the tuple. An entry in a Tuple Descriptor Table (TDT), held within the IFS/2, contains information about each set of tuples. A tuple-set is referred to by a class-number, which is used by the IFS/2 to access the TDT information.

Tuples and tuple-sets are held by the IFS/2 in its Associative Tuple Store (ATS). This consists physically of disc(s) and cache, the whole being integrated into a one-level memory by a 'paging' technique known as semantic caching [4]. The ATS cache section, which is implemented as many SIMD-parallel search engines, is also used to implement the relational and graph operations. The present IFS/2 prototype hardware unit contains 27 Megabytes of associative cache, backed by 2.2 Gigabytes of associatively-accessed disc [2]. The design is modularly extensible.
The first version of the IFS/2 software simulator (SIERRA) consists of 21 C procedures, grouped under the following headings:

- Housekeeping - (eg 'open', 'close', etc.), 5 procedures
- Search, insert, delete,
- Tuple descriptor management,
- Lexical Token Conversion - (for ASCII strings, etc.),
- Label management - (for fast dereference, etc.),

These 21 procedures are described in [3]. The following Sections introduce the procedures comprising the IFS/2’s relational algebraic command interface. Further procedures to deal with graph operations, such as transitive closure and shortest path, will be added in due course.

3. Relational processing in the IFS/2

The database concept of a relation is implemented in the IFS as that of a class, a class being a set of tuples associated with a common entry in the TDT. A relational operation on a stored class will in general be implemented as three separate low-level IFS operations:

1. Searching the ATS to retrieve the tuples in the operand;
2. performing the operation; and
3. placing the resulting tuples in the ATS for further processing.

It is reasonable to suppose that searching the ATS either to retrieve or insert tuples will account for a significant part of the time spent in performing a given relational operation. For the sake of efficiency we therefore aim to reduce as far as possible the number of separate IFS searches needed to process a complex relational query.

To this end we introduce the concept of a filter. A filter in IFS terminology is a combined selection and extended projection operation, specified by a selection criterion \( \Phi \) and \( n \) projections \( f_1, f_2, \ldots, f_n \), and transforming a relation \( R \) into the relation

\[ \{ <f_1(r), f_2(r), \ldots, f_n(r)> \mid r \in R \land \Phi(r) \} . \]

The IFS presentation of the relational algebra allows a filter to be attached to each relational operation. This helps to achieve the aim stated above by eliminating for the most part the need for separate selection and projection operations.

The operations of the algebra are chosen with relational database applications in mind: they are set union, intersection and difference, together with the extended Cartesian product and Codd’s division operator. For completeness, we give the definitions of these operations. Let \( R \) and \( S \) be relations. Then:

\[
\begin{align*}
R \cup S &= \{ r \mid r \in R \lor r \in S \} \quad \text{(set union)} \\
R \cap S &= \{ r \mid r \in R \land r \in S \} \quad \text{(set intersection)} \\
R - S &= \{ r \mid r \in R \land r \notin S \} \quad \text{(set difference)} \\
R \times S &= \{ r^0s \mid r \in R \land s \in S \} \quad \text{(extended product: \( ^0 \) denotes concatenation)} \\
R / S &= \{ p \mid (\forall s \in S)(p^0s \in R) \} \quad \text{(division)}
\end{align*}
\]

Details of the five IFS C procedure calls implementing these operations are given in Section 3.
In addition, Section 4 describes a facility provided for performing certain aggregate operations. These six primitives together comprise the IFS/2 relational interface.

4. Representing expressions

To implement the relational operations described above, we need some mechanism by which expressions may be represented and passed as parameters to procedures. The header file "ifs.h" supplied with the SIERRA package includes the following definition:

```c
struct expr
{
    node_type type;
    int     op,
            value;
    Expr    *first_arg,
             *second_arg;
};
#define Expr struct expr
```

The SIERRA relational procedures each expect to be passed an array of objects of type Expr. Each of the objects is assumed to be the head of a tree structure representing an expression which, in the presence of a given tuple, may be evaluated resulting in a single numeric or boolean value. The first expression passed is assumed to give a boolean value, and is used as a selection criterion; the rest are assumed to give numeric values, and are used as projections.

The components of an Expr object are interpreted as follows. The component node_type determines whether it represents a constant, a field number or a compound expression. For a simple expression, the field value determines the particular constant or field number. For a compound, the field op determines the particular operator, and the fields first_arg and second_arg give its arguments.

The constants recognized are the integers, together with the boolean values FALSE and TRUE, which are defined to be 0 and 1 respectively. The operators recognized are PLUS, MINUS, MULT and DIV for arithmetic; EQ, NE, GT, LE, LT and GE for comparisons; and the logical connectives AND, OR and XOR. All these names are made available by "ifs.h".

Note that some of the fields of the Expr object used to represent any particular expression may be unused.

Example: the expression which is usually written ((#2 > 3) ∧ (#4 ≠ #5)), where the ' # ' symbol indicates a field number, may be represented by the structure:
5. Relational algebraic operations

The operations of the IFS relational algebra are implemented in SIERRA by five procedures, whose declarations (in "ifs.h") take the form

IFS_status ifs_filter (IFS_class_type, int, Expr *,
                   IFS_class_type *);

IFS_status ifs_filter_union (IFS_class_type,
                   IFS_class_type, int, Expr *, IFS_class_type *);
IFS_status ifs_filter_inter (IFS_class_type,
                   IFS_class_type, int, Expr *, IFS_class_type *);
IFS_status ifs_filter_diff (IFS_class_type,
                   IFS_class_type, int, Expr *, IFS_class_type *);

IFS_status ifs_filter_prod (IFS_class_type,
                   IFS_class_type, int, Expr *, IFS_class_type *);
IFS_status ifs_filter_div (IFS_class_type,
                   IFS_class_type, int, Expr *, IFS_class_type *);

ifs_filter() implements a combined selection and extended projection, as described in Section 1, applied to a stored class. It has four parameters: the number of the stored class; the number of projections; the address of an array of Expr objects representing the selection criterion and projections; and an address at which to place the number of the new class formed.

If any of the expressions passed to the procedure refers to a nonexistent field of the stored class, the error FIELD_NO_TOO_BIG, defined in "ifs.h", is reported. When an error occurs, 0 is returned in place of a new class number.

The other procedures implement binary relational operations, so must be passed two class numbers; their other parameters are as for ifs_filter(). Note that relational joins are special uses of the ifs_filter_prod() command.

See Section 7. for an illustration of the relational interface in use.
6. Aggregate operations

The usual aggregate operations (summing, averaging, and calculating minimum and maximum values) over a given column of a given relation are supported via the procedure \texttt{ifs\_calc()}. The implementation of \texttt{ifs\_calc()} is based on the search operation; a specified aggregate operation is applied to the set of tuples responding to a specified query, and the result of that operation is returned in place of the responders. "ifs.h" includes this declaration:

\[
\text{IFS\_status ifs\_calc (IFS\_match\_algorithm, IFS\_compare\_code,}
\text{IFS\_class\_type, IFS\_query\_type, IFS\_calc\_type, short,}
\text{unsigned long *)};
\]

The first four arguments specify a search, as defined in [3]; the other three arguments specify, respectively, the aggregate operation to be performed, the field it is to be performed over, and an address at which to place the result. \texttt{IFS\_calc\_type} is an enumerated type also defined in "ifs.h", providing the names SUM, MAX, MIN and AVG for the four aggregate operations supported.

7. Persistence and transience

SIERRA supports two different kinds of class, \textit{persistent} and \textit{transient}. They differ principally in their behaviour when passed as arguments to relational operations. Transient classes are automatically removed from the tuple store (i.e., undeclared); persistent classes are not. Transient classes are intended for use in holding temporary results formed in the course of evaluating a complex query.

Classes formed by relational procedures are transient. A transient class may be made persistent by passing its number to the procedure \texttt{ifs\_persist()}; conversely, a persistent class may be made transient by calling \texttt{ifs\_unpersist()}.  

It is an error to try to insert tuples into or delete tuples from a transient class, and doing so will result in the error \texttt{CLASS\_NOT\_PERSIST} (defined in "ifs\_constants.h") being reported. However, it is possible to search a transient class, and the class searched will \textit{not} be undeclared.

8. Execution times

The elapsed time for the SIERRA simulator operations are naturally much longer than those for the actual IFS/2 hardware. In the case of relational algebraic primitives, SIERRA commands may typically be up to a thousand times slower than the corresponding hardware commands. Figure 1 shows the results of simple join tests which compare the prototype IFS/2 hardware with the following software systems:

(a) the SIERRA simulator (written in C)
(b) a hand-coded C program written specially to optimise the join command
(c) the INGRES relational DBMS
(d) Quintus PROLOG, with join fields chosen to favour Quintus's indexing strategy;
(e) Quintus PROLOG, with another combination of join fields.

All the software systems were normalised to a technology roughly comparable with that of the IFS/2's hardware, the nearest easy equivalent being a Sun/SPARC running at a clock rate of 24
MHz. (The IFS/2 hardware was configured with 27 search modules, each of 1Mbyte CAM.)

For the tests, two equally-sized relations of various cardinalities were joined. Each relation's arity was 3, each field being a 32-bit integer. The integer values were chosen so that the output relation's cardinality after the join was about 10 percent of the input relations' cardinality.

Test 1: \[ R \times S \quad 3 = 1 \]

Test 2: \[ R \times S \quad 3 = 2 \]

The PROLOG program started by reading the test data from a file and then asserting the relations, known as seta and setb. It then used the following rule for Test1:

\[ \text{join(\_)} : - \text{seta}(A,B,C), \text{setb}(C,D,E), \text{assert(int(A,B,C,D,E))), fail.} \]

The Quintus run-time was measured for execution of this rule.

Note that the times given in Figure 1 for the IFS/2 hardware are known to be capable of improvement; the prototype design is currently being refined.

The following general observations may be made on Figure 1:

(a) For small relations, the IFS/2 hardware may not be efficient. The break-even point between IFS/2 hardware and conventional software depends on the nature of the whole-structure operation and the particular software system being evaluated. For example, Figure 1 shows that the hand-coded C program performs joins faster than the IFS/2 hardware for relations smaller than about 1000 tuples. The cross-over points for the other software systems of Figure 1 lie below 100 tuples.

(b) For the particular join tests of Figure 1, there is no advantage in having more than 27 search modules in the IFS/2 hardware for cardinalities under about 3000 tuples. Above that size, the IFS/2 performance curve can be kept linear by adding more search modules.
Figure 1: join comparisons, IFS/2 hardware versus various software systems.
9. A demonstration

The following is meant to illustrate the use of some of the IFS relational operations on a small test database. It reproduces the output of a program which prompts for an operation and its arguments, then calls the corresponding simulator function and displays the contents of the resulting class. The demonstration program begins by displaying a menu:

---
P (filter-product demo)
U (filter-union demo)
I (filter-intersect demo)
U (filter-difference demo)
F (filter demo)
C (calc demo)
H (how many - count demo)
S (show class)
Q (quit)
---

The database contains the batting and bowling statistics for one innings of a recent cricket match. There are two relations, one holding the batting records, and the other the bowling records.

? S
? Table no: 1

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Wright</td>
<td>b</td>
<td>Lewis</td>
<td>6</td>
</tr>
<tr>
<td>Latham</td>
<td>lbw</td>
<td>Pringle_D</td>
<td>25</td>
</tr>
<tr>
<td>Crowe</td>
<td>b</td>
<td>Reeves</td>
<td>31</td>
</tr>
<tr>
<td>Jones</td>
<td>c</td>
<td>Reeves</td>
<td>1</td>
</tr>
<tr>
<td>Greatbatch</td>
<td>c</td>
<td>Reeves</td>
<td>4</td>
</tr>
<tr>
<td>Harris</td>
<td>not_out</td>
<td>-</td>
<td>38</td>
</tr>
<tr>
<td>Cairns</td>
<td>c</td>
<td>Pringle_D</td>
<td>42</td>
</tr>
<tr>
<td>Smith</td>
<td>c</td>
<td>Lewis</td>
<td>2</td>
</tr>
<tr>
<td>Pringle_C</td>
<td>not_out</td>
<td>-</td>
<td>9</td>
</tr>
</tbody>
</table>

? S
? Table no: 2

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>De_Freitas</td>
<td>10</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>Lewis</td>
<td>8</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>Pringle_D</td>
<td>6</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>Reeves</td>
<td>10</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Tufnell</td>
<td>10</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>Hick</td>
<td>6</td>
<td>0</td>
<td>29</td>
</tr>
</tbody>
</table>
To get the names of those batsmen scoring more than twenty runs, together with their scores, we perform a filter() on the first relation:

? F

? Table no: 1
? Select-EXPR: (#4>20)
? No of output fields: 2
? Field no 1: #1
? Field no 2: #4

class number  3

    < Latham, 25 >
    < Crowe, 31 >
    < Harris, 38 >
    < Cairns, 42 >

By using a more complex selection condition, we can include those batsmen who were not out.
(The symbol '|' in the selection condition stands for logical-or):

? F

? Table no: 1
? Select-EXPR: ((#4>20)|(#2="not_out"))
? No of output fields: 2
? Field no 1: #1
? Field no 2: #4

new table is    #4
class number  4

    < Latham, 25 >
    < Crowe, 31 >
    < Harris, 38 >
    < Cairns, 42 >
    < Pringle_C, 9 >

To illustrate a join query, we can use filter_prod() to form a table in which the records of dismissed batsmen each have appended to them the record of the bowler concerned:

? P

? Table no 1: 1

? Table no 2: 2

? Select-EXPR: (#3=#5)
? No of output fields: 8
? Field no 1: #1
class number 5

< Wright, b, 6, Lewis, 8, 0, 33, 2 >
< Latham, lbw, 25, Pringle_D, 6, 1, 31, 2 >
< Crowe, b, 31, Reeve, 10, 3, 20, 3 >
< Jones, c, 1, Reeve, 10, 3, 20, 3 >
< Greatbatch, c, 4, Reeve, 10, 3, 20, 3 >
< Cairns, c, 42, Pringle_D, 6, 1, 31, 2 >
< Smith, c, 2, Lewis, 8, 0, 33, 2 >

Finally, to illustrate the use of ifs_calc(), we find the average of the batsmen's scores. The program automatically fills in an all-wild search pattern.

? C

? Table no: 1
? S(sum) M(Max) L(east) A(vg) = A
? Col no = 4

Result is 17

? Q
8. Acknowledgements

It is a pleasure to acknowledge the contribution of other members of the IFS team at Essex. In particular, Andy Marsh and Chetan Mistry helped compile the statistics for Figure 1. The work described in this report has been supported by SERC grants GR/F/61028 and GR/G/30867.

9. References


